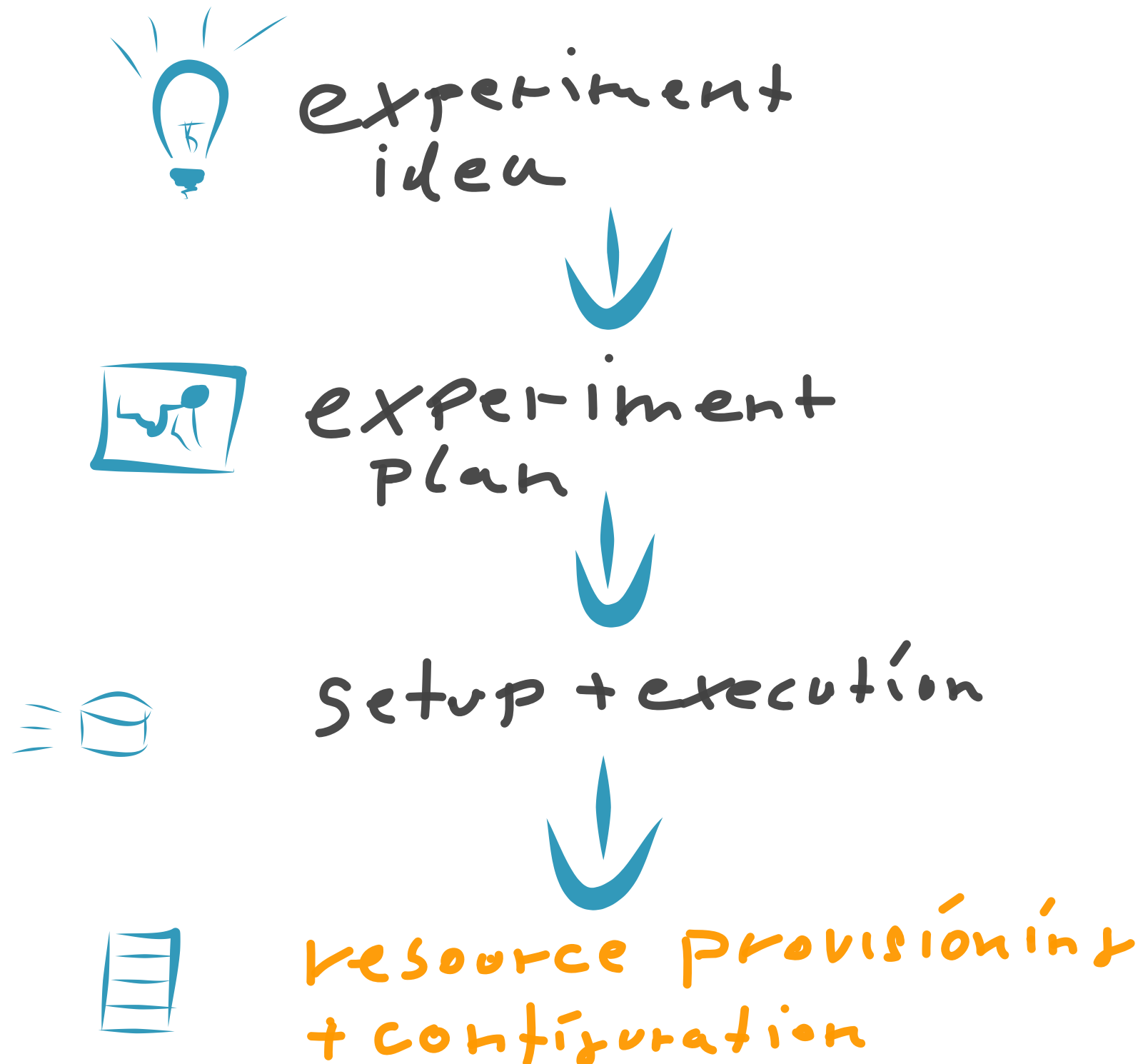


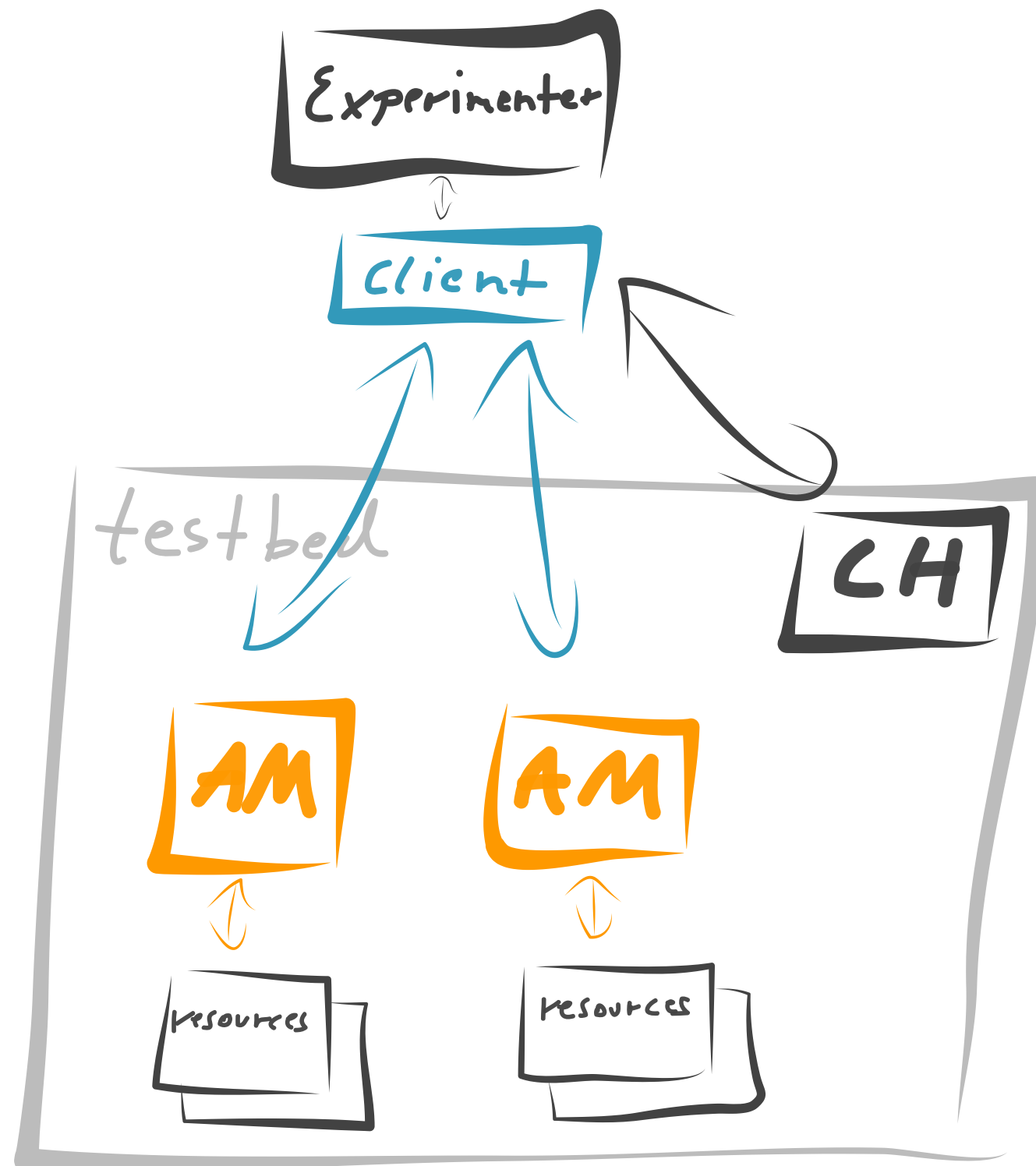
# AMsoil

The glue for Aggregate Manager developers

# researcher's goal



# experiment execution

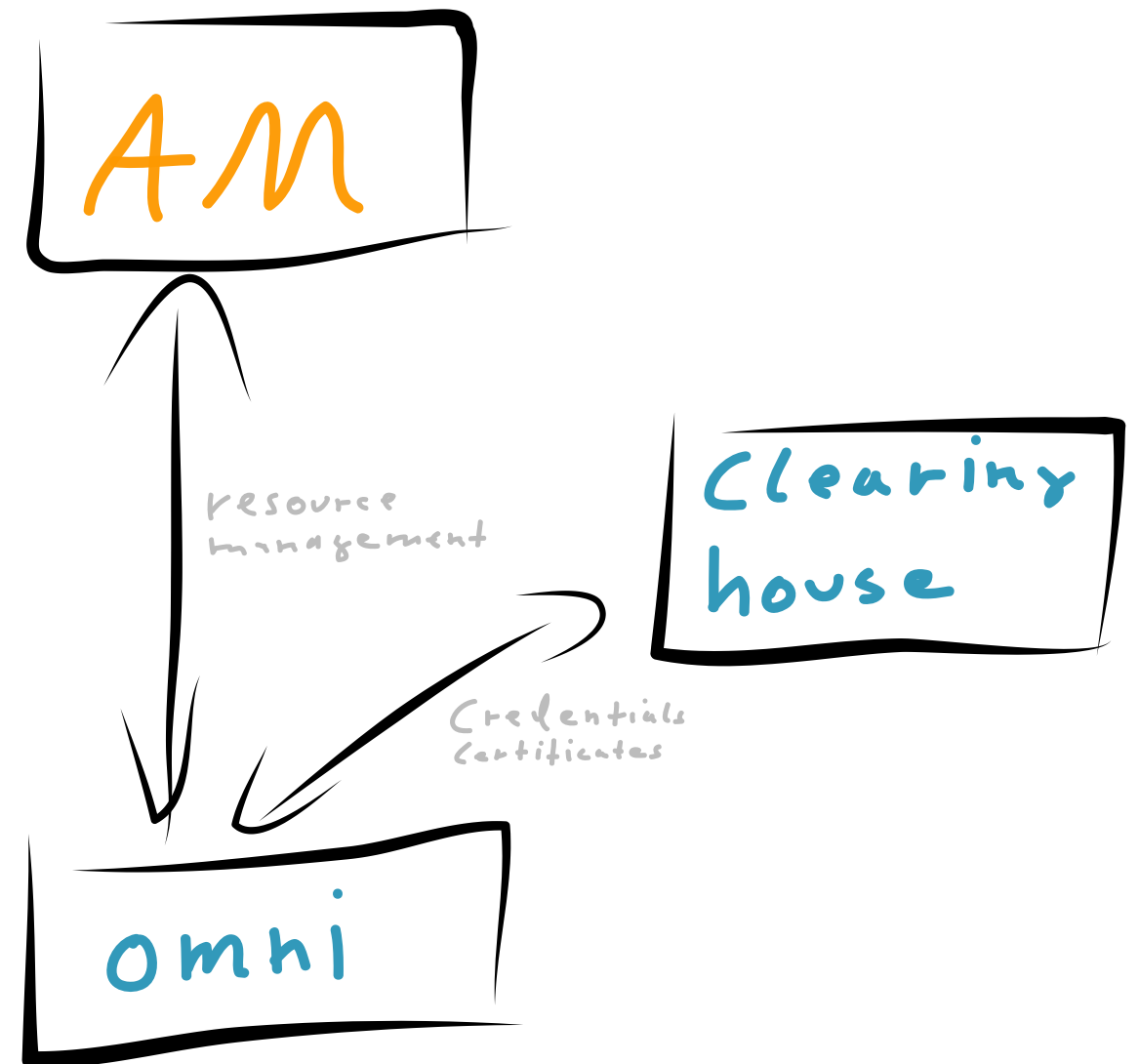


CH Clearinghouse

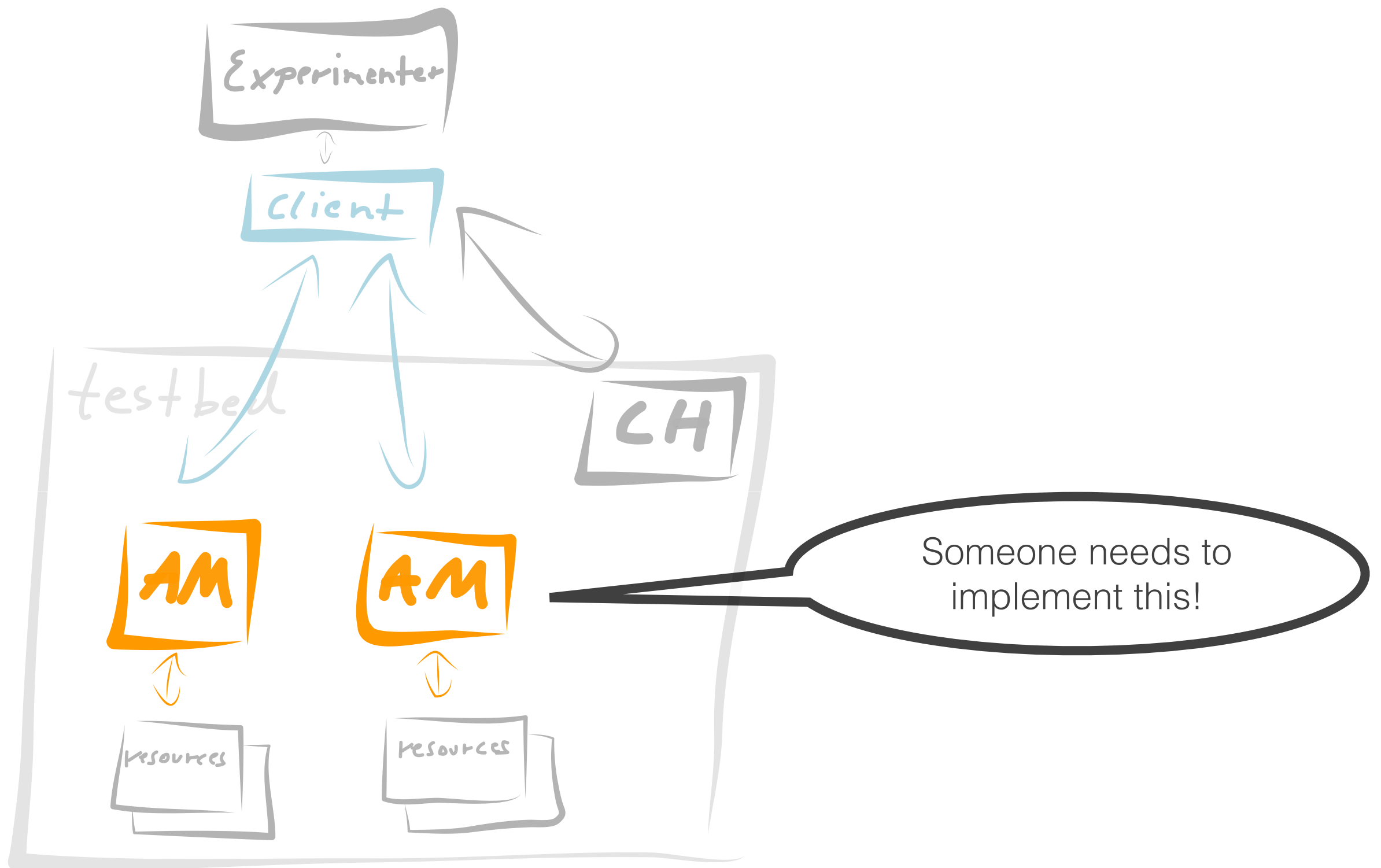
AM Aggregate Manager

# test bed

- **Clearinghouse** manages certificates and credentials
- The **client** (*here: omni*) assembles the request and sends it to the Aggregate Manager
- **Aggregate Manager** manages, allocates and provisions resources



# AMsoil?



# AMsoil?

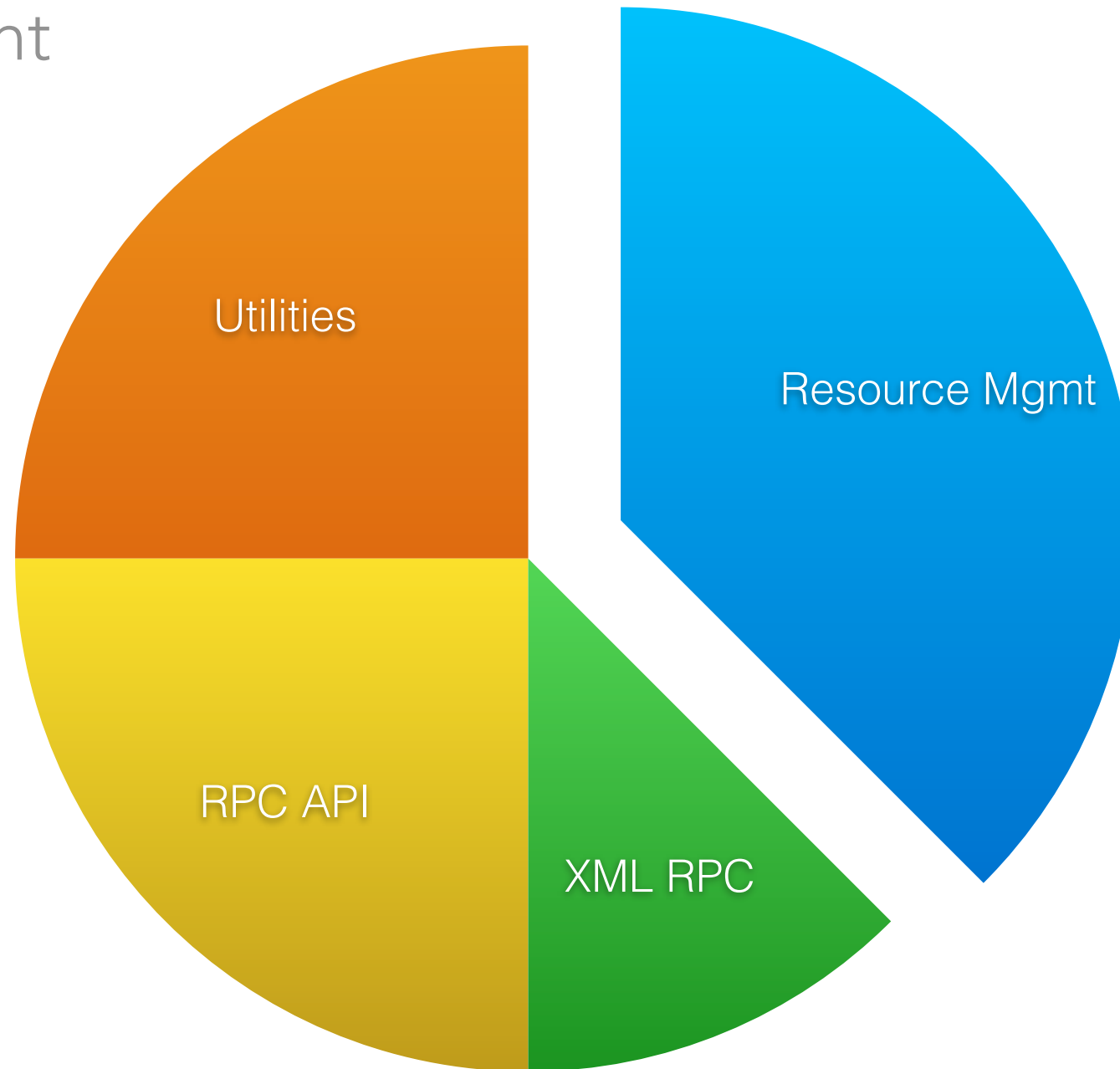


AMsoil is a light-weight [framework for creating Aggregate Managers](#) in test beds.

AMsoil is a pluggable system and provides the necessary glue between RPC-Handlers and Resource Managers . Also it provides helpers for common tasks in [AM development](#).

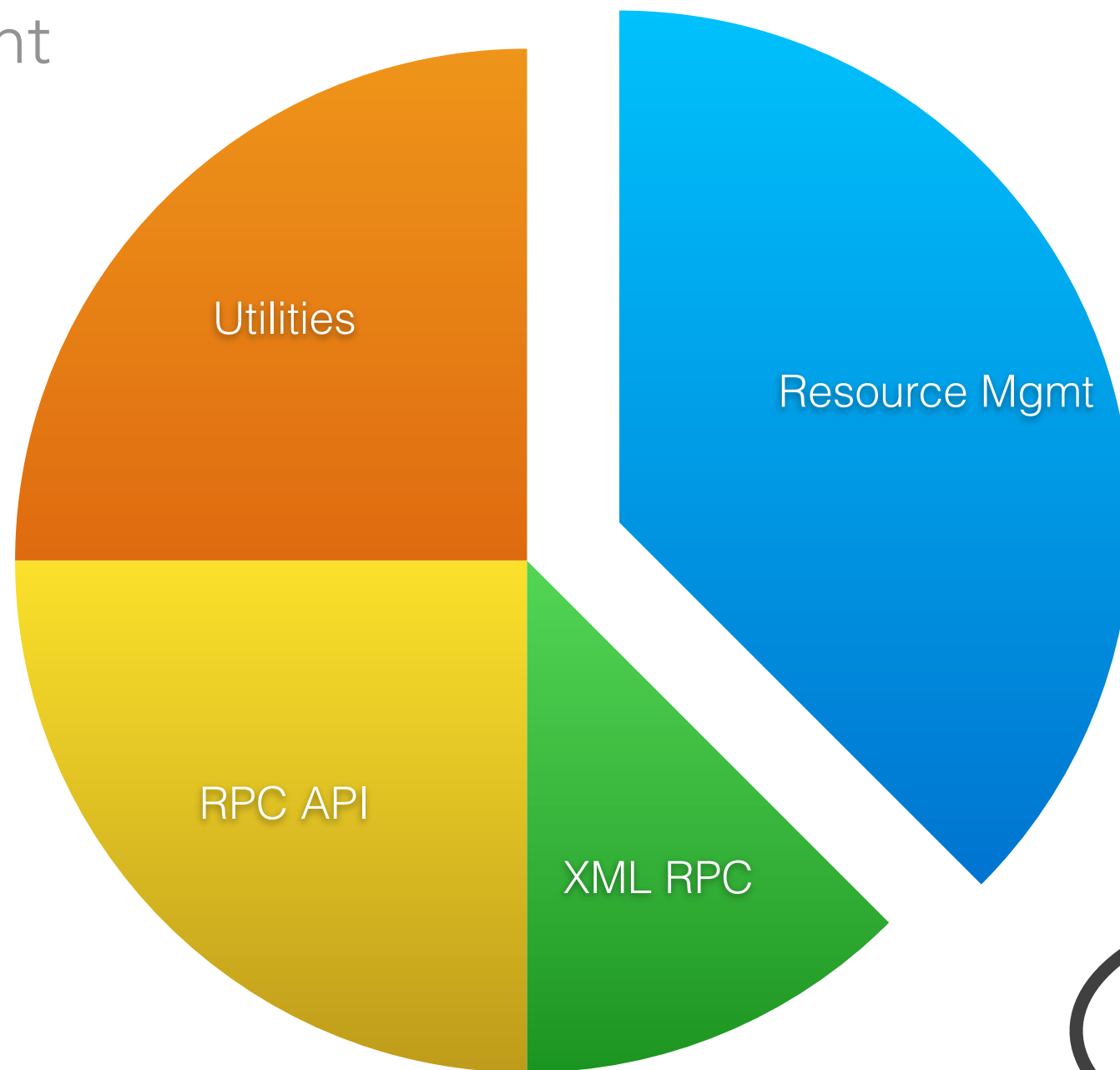
# motivation

AM development  
without AMsoil



# motivation

AM development  
without AMsoil

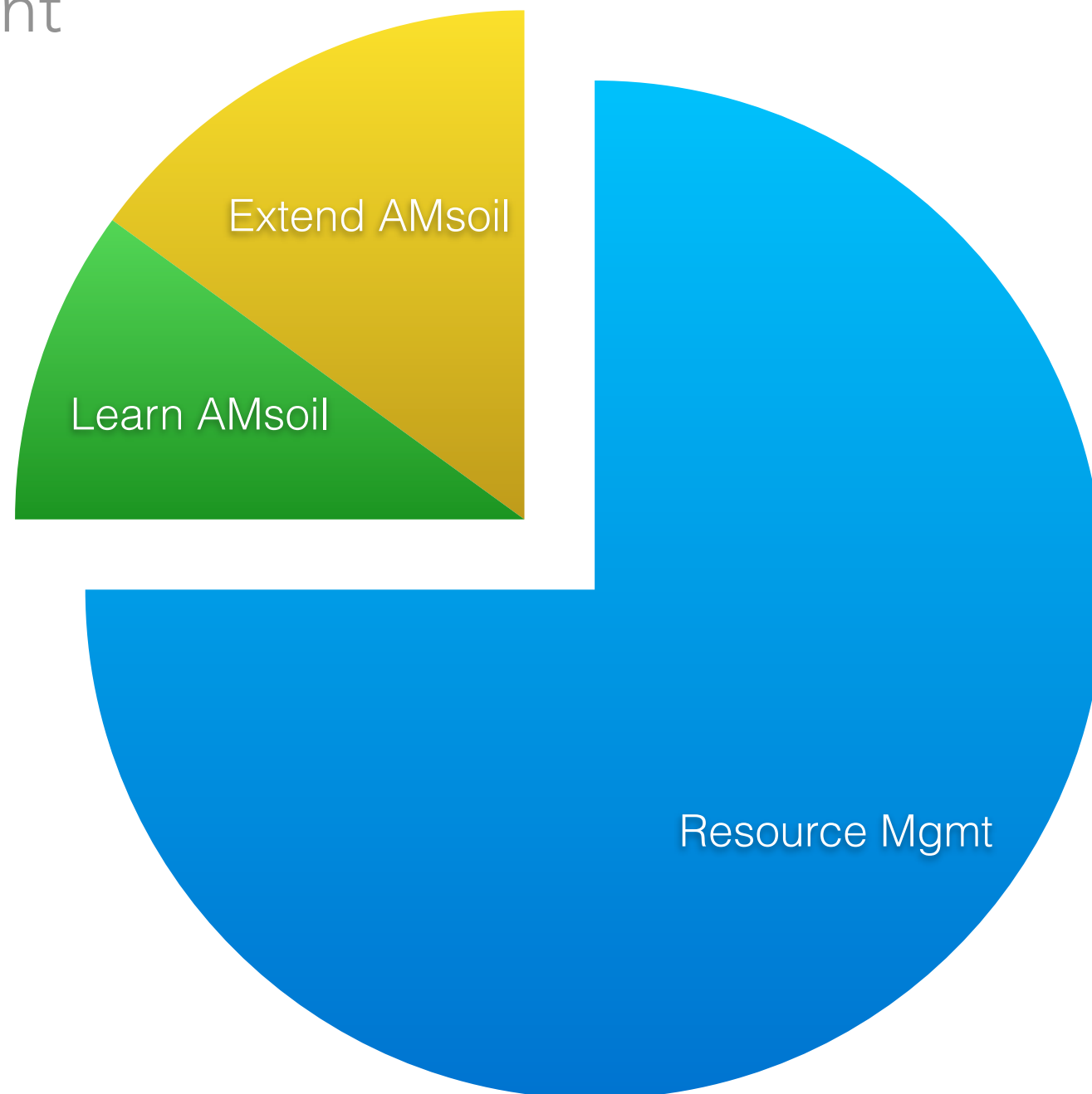


This is why  
you write an AM.  
The rest is just annoying.



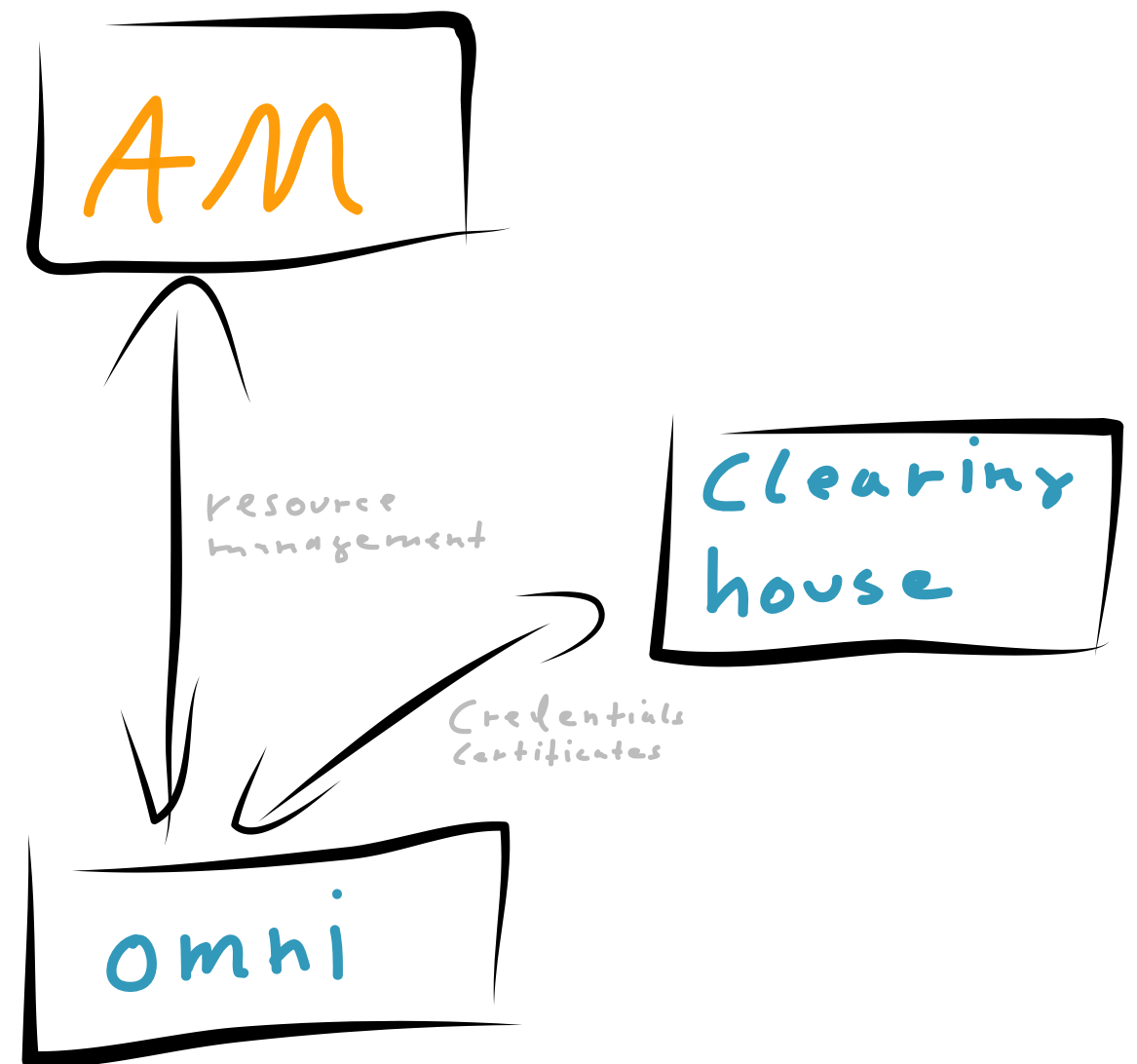
# motivation

AM development  
with AMsoil



# how to write an AM

- Setup a little test bed
  - Install a Clearinghouse
  - Install a client
  - Install AMsoil
- Understand AMsoil
- Start hacking...



# what now?

finish this presentation,

**clone** the repository [↗ https://github.com/fp7-ofelia/AMsoil.git](https://github.com/fp7-ofelia/AMsoil.git)

then **read** [↗ https://github.com/fp7-ofelia/AMsoil/wiki/Installation](https://github.com/fp7-ofelia/AMsoil/wiki/Installation)

# what you need to know

- how [plugins](#) work
- how a [GENI](#) testbed works
- what plugins you [need to develop](#)
- what else [AMsoil](#) supports

# a broad look

## AMsoil's directory structure

```
|-- admin
|-- deploy
|   |-- trusted
|-- doc                                Documentation
|   |-- img
|   |-- wiki
|-- log                                AMsoil's log
|-- src
|   |-- amsoil                         AMsoil's core implementation
|       |-- core
|   |-- disabled-plugins              Unused code/plugins
|       |-- ...
|   |-- plugins                       Plugins to be loaded when bootstrapping AMsoil
|       |-- ...
|-- test
```

# why plugins?

- **Selection**

An administrator can add/remove plugins/functionality.

- **Exchangeability**

The interface remains, but the implementation be changed.

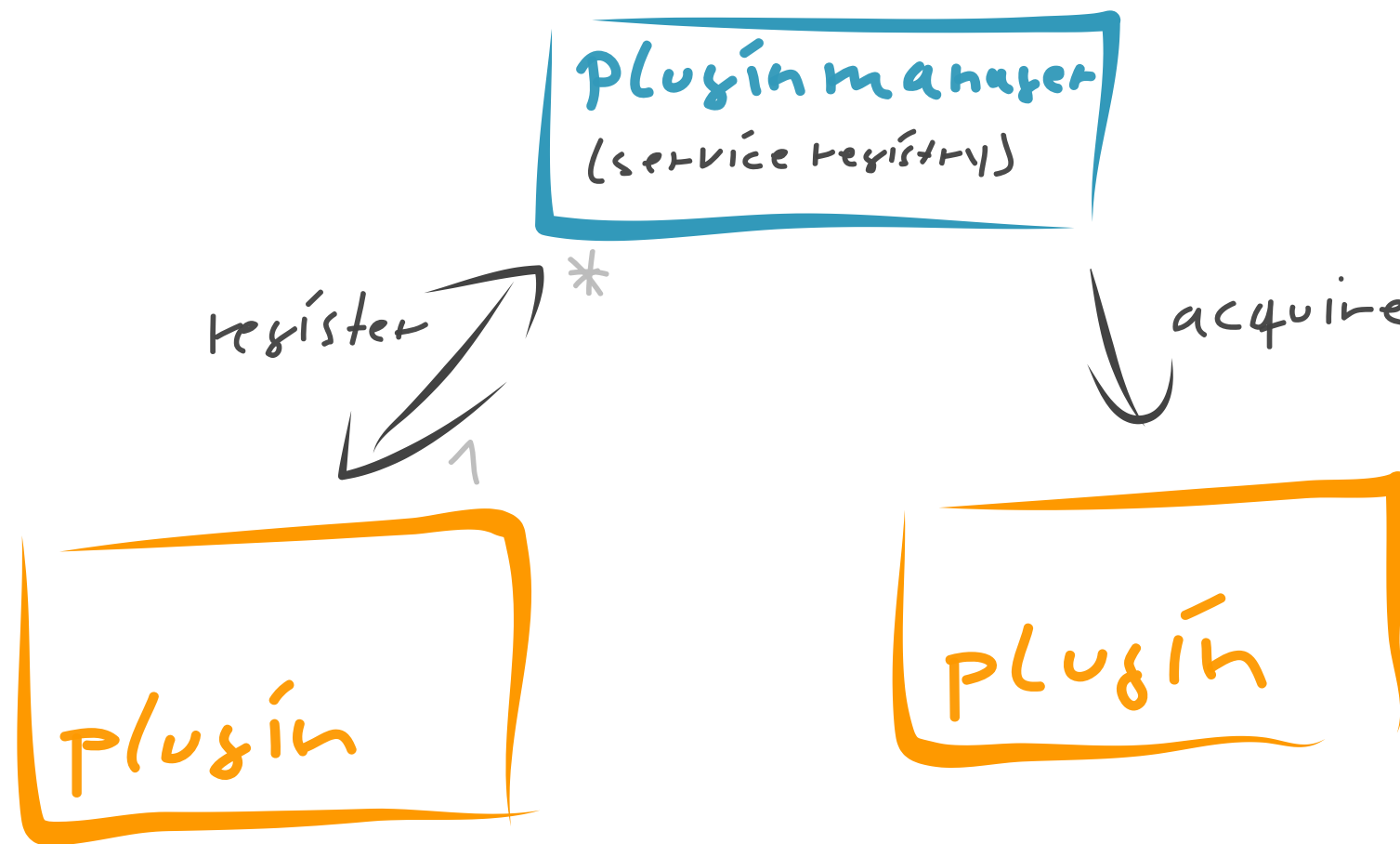
- **Clarity**

Provide a set of services and hide the details behind.

- **Encapsulation**

Protect implementations from other developers.

# register and use plugins



```
[plugin A] import amsoil.core.pluginmanager as pm
[plugin A] pm.registerService('myservice', serviceObject)

[plugin B] service = pm.getService('worker')
[plugin B] service.do_something(123)
```


# what can be a service?

short version

**everything** which can be referenced in Python

long version

ints, strings, lists, dicts, objects, classes, packages



yes even  
packages!



# under the hood

## plugin

describes  
services & dependencies

Manifest

performs  
initialization & registration

plugin.py

implementation

# implement a plugin

- create a [new folder](#) in plugins
- create the [manifest.json](#)
- create the [plugin.py](#)
  - write a [setup\(\)](#) method
- [register](#) your services

# implement a plugin

manifest.json

```
{
  "name"          : "My Plugin Name",
  "author"        : "Tom Rothe",
  "author-email"  : "tom.rothe@eict.de",
  "version"       : 1,
  "implements"    : ["myservice", "myclass", "mypackage"], # you'll register these services
  "loads-after"   : ["somedependency"],                  # dependency needs to be loaded before the setup method
  "requires"      : []                                    # dependency can be loaded after the setup method
}
```

plugin.py

```
# ...imports...
def setup():
    # register a service
    pm.registerService('myclass', ServiceClass)
    pm.registerService('myinstance', SingleClass() )
    pm.registerService('mypackage', my.python.package)
```

# @serviceinterface

The methods and attributes which can should be used are marked the annotation @serviceinterface.

## implementation

```
from amsoil.core import serviceinterface

class MyService(object):
    @serviceinterface
    def do_something(self, param):      # can be used by the service user
        pass
    def do_more(self, param):          # not part of the service contract, NOT to be used
        pass
```

# DOs and DONTs

- If you have plugin-specific exceptions, create a [package with all exceptions](#) and register the package as a service.
- Separate a plugin [into multiple plugins](#) if this improves re-usability.
- [Never import another plugin directly](#), always go via the pluginmanager via `pm.getService()`.

# now what?

AMsoil managers are used in a GENI-like test bed.

Let's understand how GENI works.

# names in GENI

- **Experimenter**

A human user who uses a client to manage resources via an AM.

- **Sliver**

A physical or virtual resource. It is the smallest entity which can be addressed by an AM

(e.g. an IP address, a virtual machine, a FlowSpace).

- **Slice**

A collection of slivers.



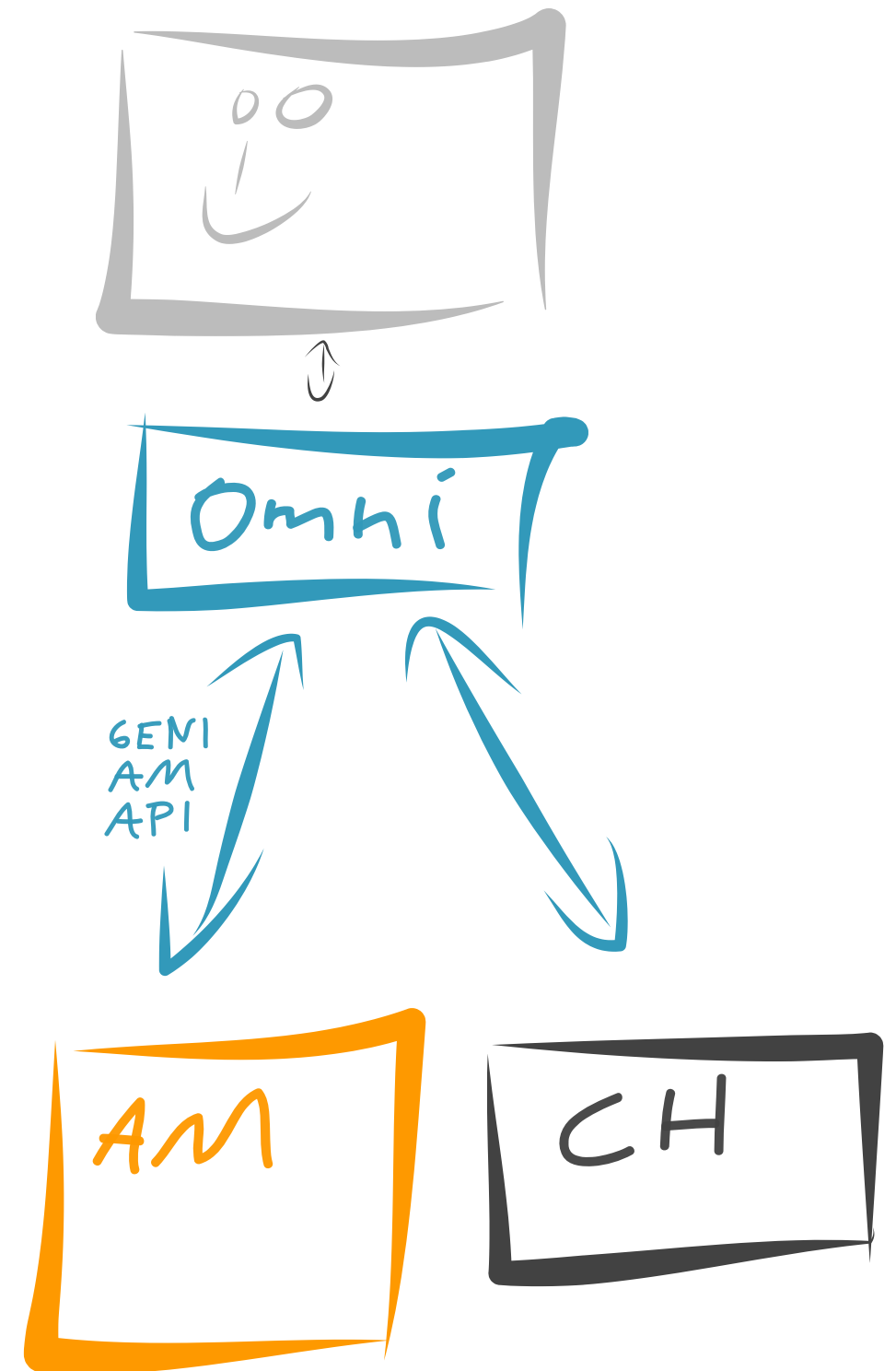
# communication

- The Clearinghouse provides services to know who you are and what you may do.

*(we don't care, just use it)*

- The client speaks the GENI AM API to the AM.

*(we care, because we implement it)*

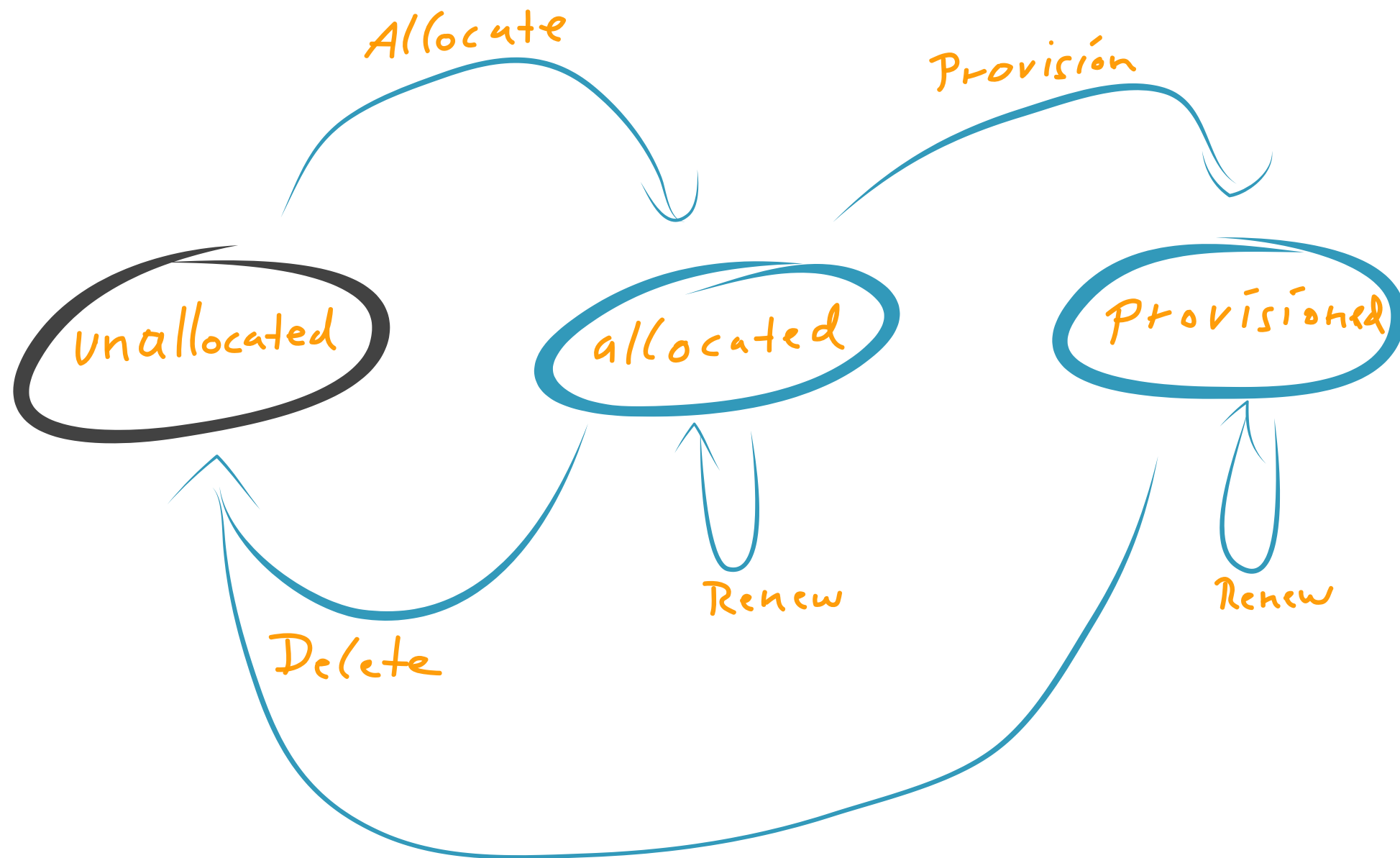




# what can the API do?

<code>GetVersion</code>	Get info about the AM's
<code>ListResources</code>	Info what the AM has to offer
<code>Describe</code>	Info for a sliver
<code>Allocate</code>	Reserve a slice/sliver for a short time
<code>Renew</code>	Extend the usage of a slice/sliver
<code>Provision</code>	Provision a reservation for a longer time
<code>Status</code>	Get the status of a sliver
<code>PerformOperationalAction</code>	Change the operational state of a sliver
<code>Delete</code>	Remove a slice/sliver
<code>Shutdown</code>	Emergency stop a slice

# allocate *and* provision?



allocated only for a short time resources are only booked not provisioned  
provisioned the slice/sliver actually takes up resources (is actually usable)

# typical experiment

*Imagine a restaurant reservation.*

- **ListResources**

Call the restaurant to ask what tables are available.

- **Allocate**

Call to tell which table you want (they will only hold the table for 2 hours).

- **Provision**

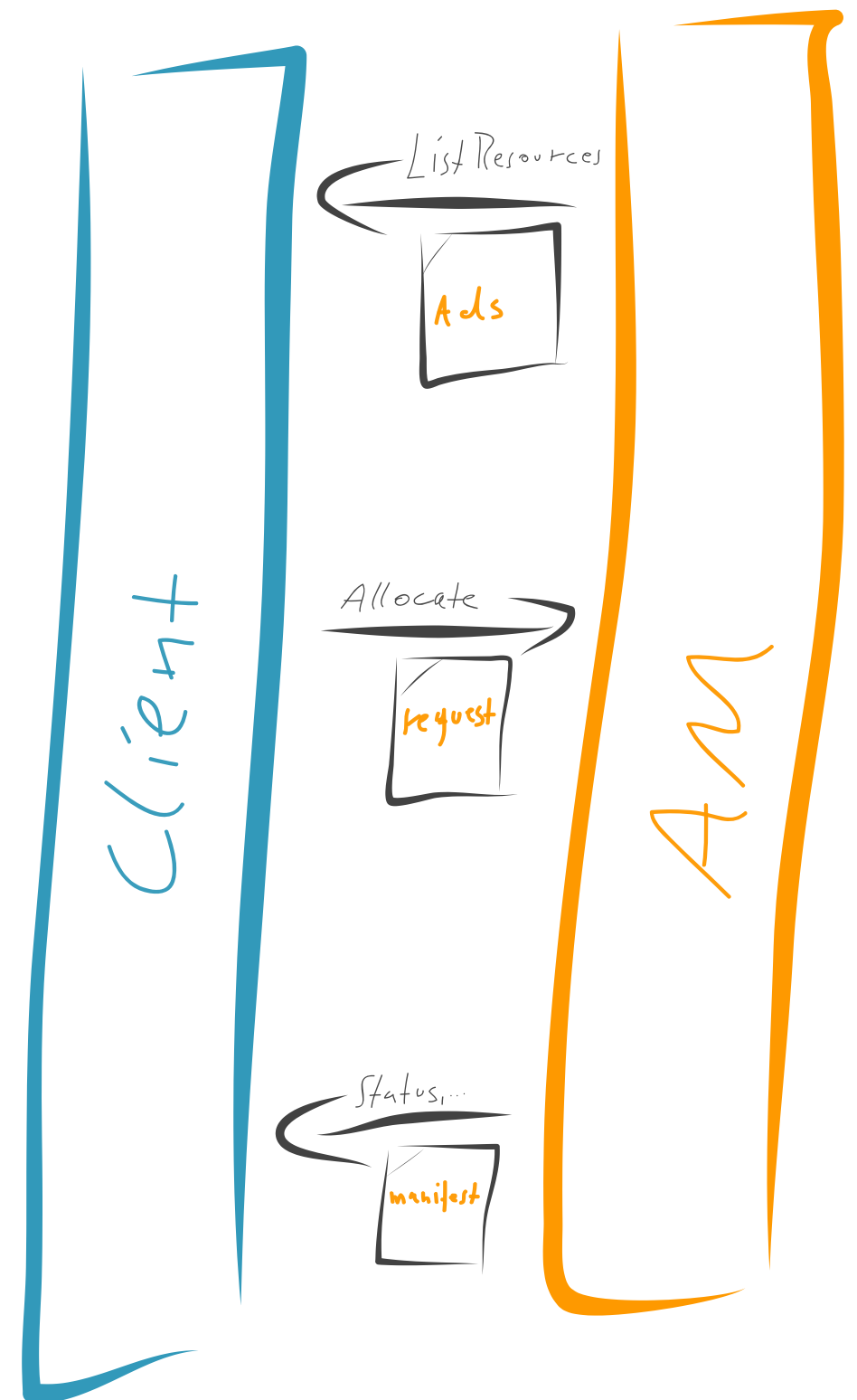
Come and use at the table (this may take 5 hours).

# how do say what I want?

The resources are described with an XML document called RSpec.

There are three RSpec types:

- **Advertisement** *(short: ads)*  
Announces which resources/slivers are available.
- **Request**  
Specifies the wishes of the experimenter
- **Manifest**  
Shows the status of a sliver



# enough theory!

Now we know what plugins are and how GENI works.

Let's put it together and write an AM.

# getting the requests

- **RPC Handler**

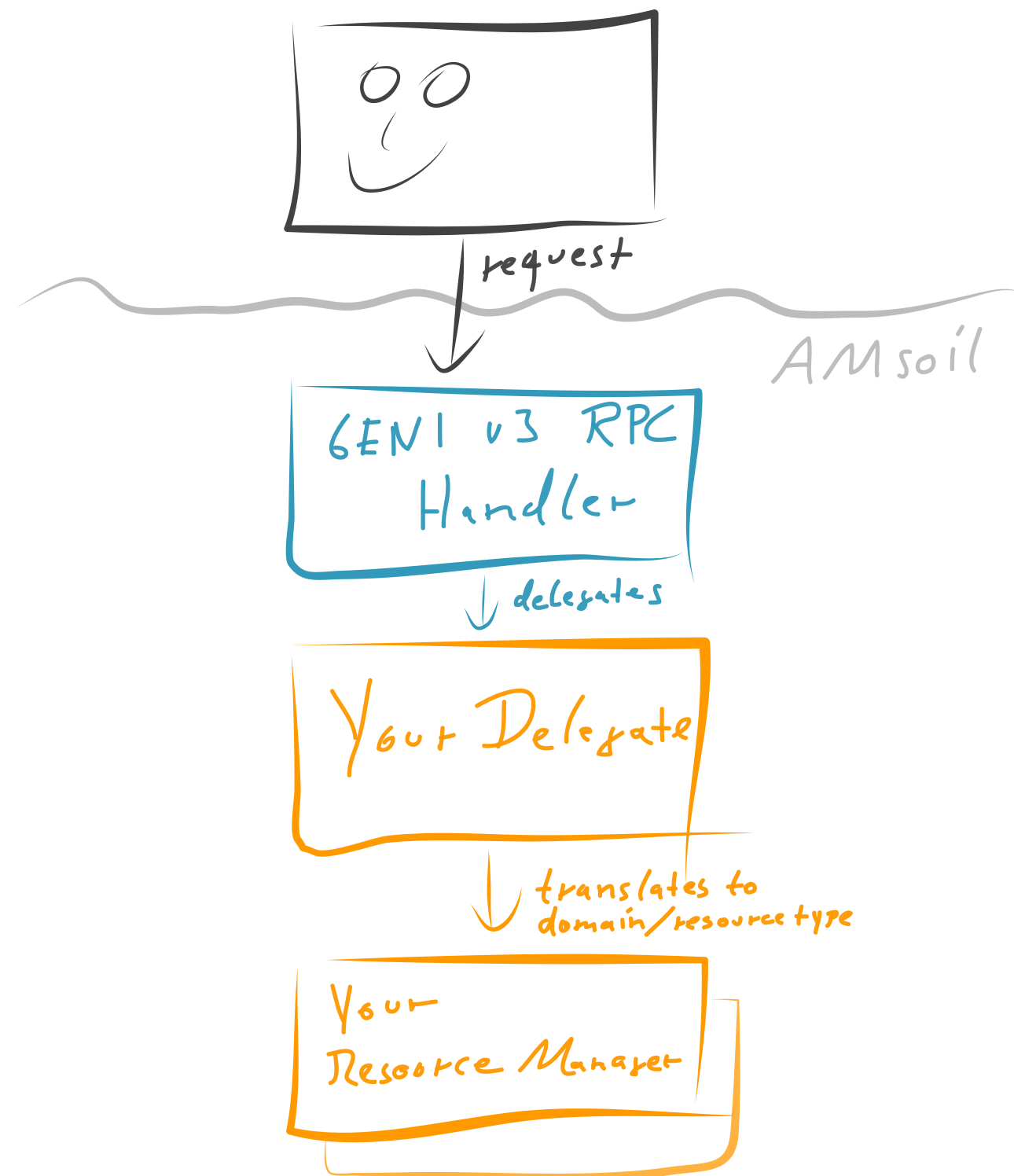
Retrieves the XML-RPC request, does some magic and passes the request on to the delegate.

- **Delegate**

Translates the GENI request into a language the Resource Manager can understand

- **Resource Manager** (short: RM)

Handles the actual allocation of the resources.



# why RM *and* Delegate?



We need to **decouple** the RPC API from the resource management logic.

This enables AMsoil-based AMs to implement **multiple APIs** (e.g. GENI, SFA, OFELIA APIs) without having to re-write everything.

# interfaces

- **Delegate**

Should derive from DelegateBase and overwrite the methods prescribed (e.g. list\_resources, allocate, ...).

- **Resource Manager**

You make up the interface!

The methods, attributes, parameters are domain-specific and depend on the resource type being handled.



# a new plugin is born

Create new plugins which handle the incoming requests from the client and do the actual resource management.

## YourDelegate

- ✓ New folder for plugin
- ✓ manifest.json
- ✓ plugin.py
- ✓ a delegate object

## YourResourceManager

- ✓ New folder for plugin
- ✓ manifest.json
- ✓ plugin.py
- ✓ a manager service

# YourDelegate

yourdelegate/plugin.py

```
# ...imports...
GENIv3DelegateBase = pm.getService('geniv3delegatebase')
geni_ex = pm.getService('geniv3exceptions')

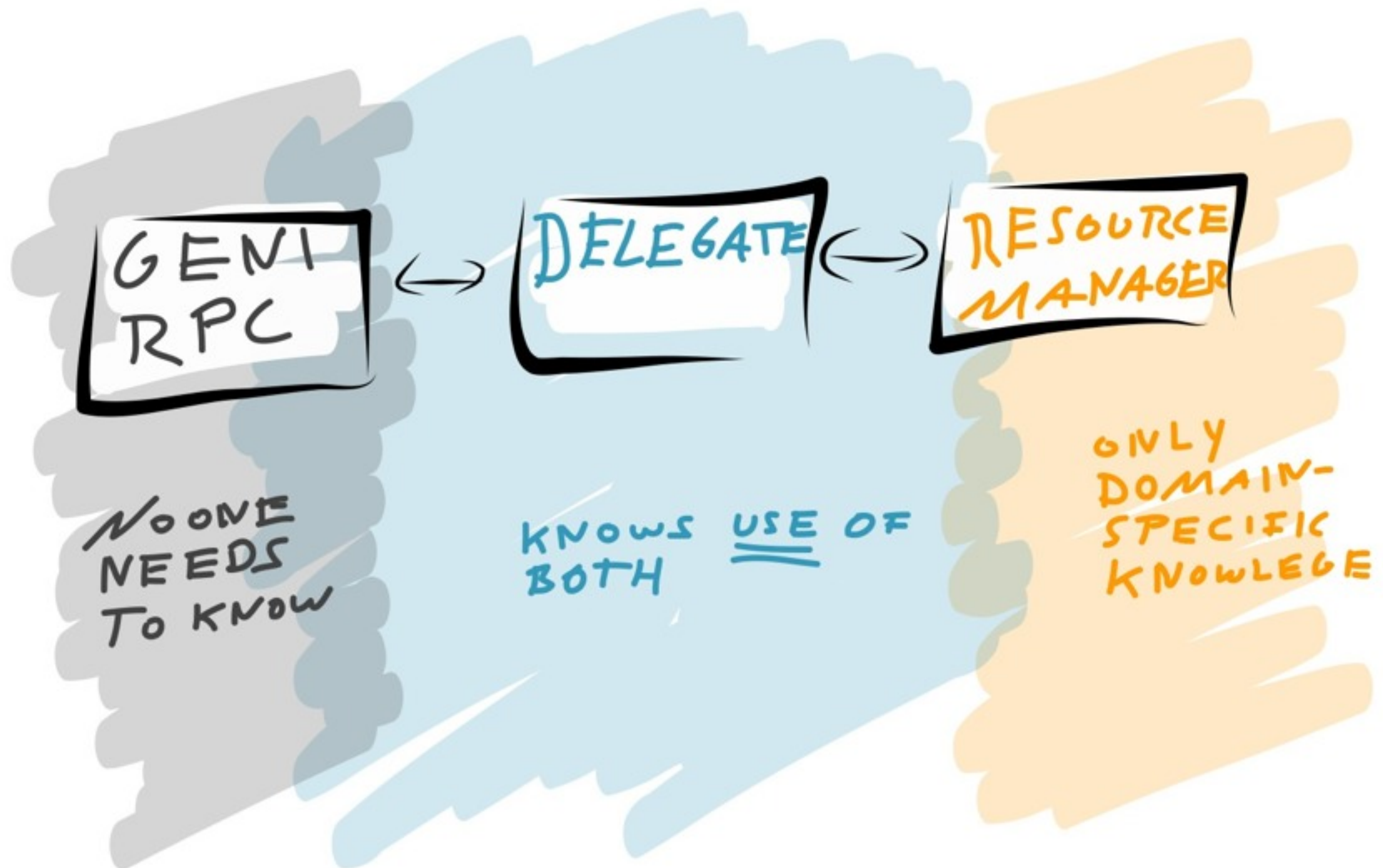
class MyDelegate(GENIv3DelegateBase): # derive from DelegateBase
    # ...
    def allocate(self, slice_urn, client_cert, credentials, rspec, end_time=None): # Overwrite DelegateBase method
        # perform authentication and check the privileges
        client_urn, client_uuid, client_email = self.auth(client_cert, credentials, slice_urn, ('createsliver',))

        rspec = self.lxml_parse_rspec(rspec) # call a helper method to parse the RSpec (incl. validation)
        # ...interpret the RSpec XML...
        try:
            # call a resource manager and make the allocation happen
            self._resource_manager.reserve_lease(id_from_rspec, slice_urn, client_uuid, client_email, end_time)
        except myresource.MyResourceNotFound as e: # translate the resource manager exceptions to GENI exceptions
            raise geni_ex.GENIv3SearchFailedError("The desired my_resource(s) could not be found.")

        return self.lxml_to_string("<xml>omitted</xml>"), {'status' : '...omitted...'} # return the required results

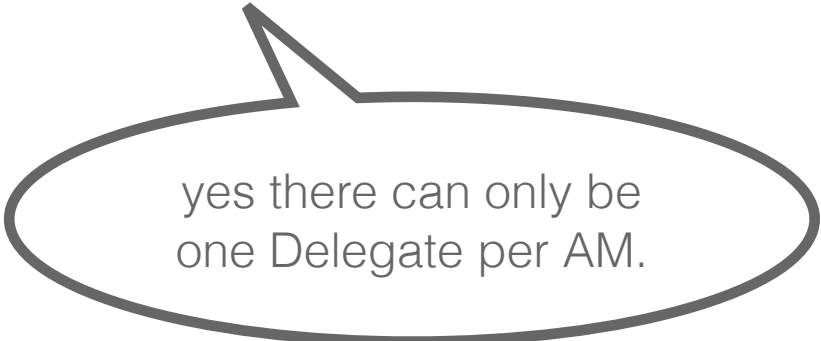
def setup():
    delegate = MyGENI3Delegate()
    handler = pm.getService('geniv3handler')
    handler.setDelegate(delegate)
```

# needed knowledge



# Delegate tasks

- Translate GENI API into Resource Manager(s) methods
- Translate the RSpecs into Resource Manager values (and back).
- Catch Resource Manager errors and re-throw as GENIv3....
- Translate the namespace from GENI to RM (e.g. URN ↔ UUIDs).
- Specify the needed privileges for authorization.
- De-multiplex to dispatch to different Resource Managers  
(if you have multiple resource types in one AM).



yes there can only be  
one Delegate per AM.

# RM tasks

- Instantiate resources
- Manage persistence of reservations and resource state
- Check policies
- Avoid collisions of resources reservations /  
Manage availability
- Throw domain-specific errors

# more info

- Please see the [wiki](#) for
  - Authentication / Authorization tools
  - RSpec generation assistance
  - More detailed description
- Checkout the code and look at the DHCP AM example
  - plugin: dhcprm
  - plugin: dhcpgeni3
  - API description of geniv3rpc

# hands on tips

Let's see how we can make our life even easier.

# testing

- ✓ Fire up the Clearinghouse
- ✓ Start the AMsoil server
- ✓ Run omni to send a request
  - ✓ Check AMsoil's logs

```
gcf#      python src/gcf-ch.py
amsoil#    python src/main.py
amsoil#    tail -f log/amsoil.log
gcf#      python src/omni.py -o -a https://localhost:8001 -V 3 getversion
```



# development mode

- Use the configuration tool to set `flask.debug = True`
  - Now the server [reloads its files every time](#) you change a file.

!! Careful: The client's certificate is now read from a pre-configured file.
- For debugging
  - Throw exceptions or
  - Write to the log to see what's going on.

# logging

anywhere.py

```
import amsoil.core.log
logger=amsoil.core.log.getLogger('pluginname')
# logger is a decorated instance of Python's logging.Logger, so we only get one instance per name.

def somemethod():
    logger.info("doing really cool stuff...")
    logger.warn("Oh Oh...")
    logger.error("Ba-Boooom!!!")
```

# configuration

anywhere.py

```
import amsoil.core.pluginmanager as pm
config = pm.getService("config")      # get the service
myvalue = config.get("mygroup.mykey") # retrieve a value
config.set("mygroup.mykey", myvalue)  # set a value
```

plugin.py

```
import amsoil.core.pluginmanager as pm
def setup():
    config = pm.getService("config") # get the service
    config.install("mygroup.mykey", "somedefault", "Some super description.") # install a config item
```



Always install the config keys and defaults on the plugin's setup method (install will not re-create/overwrite existing entries).

# worker

The worker enables dispatching jobs to an external process (e.g. to perform longer tasks without blocking the client's request response).

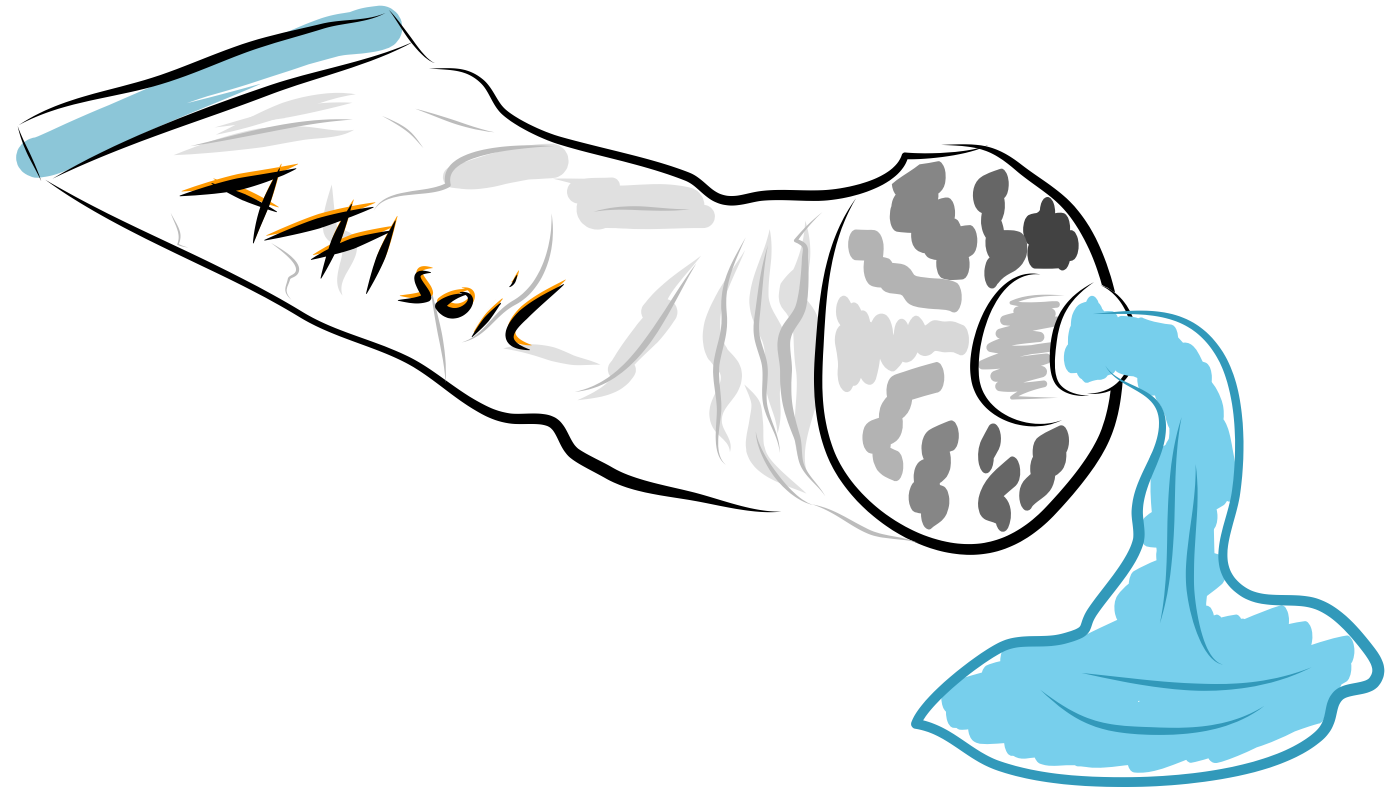
anywhere.py

```
worker = pm.getService('worker') # get the service
worker.add("myservice", "mymethod", "parameter1") # run as soon as possible
worker.addAsRecurring("myservice", "mymethod", [1,2,3], 60) # run every minute
worker.addAsScheduled("myservice", "mymethod", None, datetime.now() + timedelta(0, 60*60*2)) # run in 2 hours
```

fire up the server (needs reboot when changing code)

```
amsoil# python src/main.py --worker
```

# you know it all



clone the repository

🔗 <https://github.com/fp7-ofelia/AMsoil.git>

then read the wiki

🔗 <https://github.com/fp7-ofelia/AMsoil/wiki>